

Generalized LR parsing and the shuffle operator

John Maraist

Computer Science Department, University of Wisconsin-La Crosse,
1725 State Street, La Crosse, Wisconsin 54601, USA, jmaraist@uwlax.edu

Abstract. We adapt Tomita’s Generalized LR algorithm to languages generated by context-free grammars enriched with a shuffle operator. The change involves extensions to the underlying handle-finding finite automaton, construction of parser tables, and the necessary optimizations in constructing a deterministic parser. Our system is motivated by an application from artificial intelligence plan recognition. We argue for the correctness of the system, and discuss future extensions of this work.

1 Introduction

In this paper we study the extension of context-free grammars (CFGs) with *shuffle operations*. Shuffle operations combine strings so that the order of symbols from each string is preserved, but interleaving of the shuffled strings is possible. For example, both m12np3r45 and mn123pr45 are shufflings of mnpr and 12345, but mp12nr345 is not because the n and p occur in a different order. Although shuffling has long been an aspect of concurrent systems analysis, the study of shuffled or *intermixed* languages has lagged behind their use [9]. Work through now on intermixed languages has focused on the theoretical general properties of language classes, or on practical approaches to regular expressions and languages with shuffling [10,11].

Our interest in the shuffling operator is motivated by an application from artificial intelligence. *Plan recognition* is the problem of determining the goal (or goals) and plan of an actor from a sequence of observed actions. The connection between plan recognition and parsing is well known [14]. The requirements and structure of plan recognition and parsing algorithms differ in some significant ways. Grammars enumerate an order among all structures in a language, and parsers expect that total order; the libraries defining the plans corresponding to a goal will often give only a partial order, or no order, in some cases. Parsers are directed to the understanding of a stream of inputs corresponding to a single top-level entity; plan recognizers should be able to recognize the pursuit of multiple goals at the same time. And parsers are designed with the assumption that the entire string to be parsed is available from the start of parsing, but plan recognizers are typically expected to draw preliminary conclusions as soon as each piece of input is available.

Early approaches to plan recognition stayed close to parsing algorithms; they did not address recognition of multiple goals executed simultaneously, and they did not accept plan libraries specifying a non-total order among their steps [8,15]. Goldman, Geib and Miller's system PHATT relaxed these restrictions [3,5], and subsequent work significantly improved PHATT's performance [4,7]. Geib has also similarly adapted parsers for combinatory categorial grammars (CCGs), a more complicated representation than CFGs, into plan recognizers [2]. These approaches are all somewhat *ad hoc*, in that they develop a algorithm inspired by a CFG parsing algorithm but neither address the shuffle operator explicitly, nor identify its impact on the underlying parser. Moreover, these approaches are all fundamentally based on *top-down* parsing algorithms, producing a plan recognizer which must separately represent the different interpretations of inputs.

Our main contribution here is to extend Tomita's Generalized LR (GLR) parser [12,13] to languages generated by CFGs enriched with the shuffle operator. First, in Section 2 we formalize our notion of context-free shuffle grammars (CFSG), and define rewriting relations for CFSGs. We then introduce our GLR-S algorithm in two steps. In Section 3 we present a nondeterministic GLR-S parser, and in Section 4 we discuss how the parser can be efficiently implemented. Finally we conclude with a discussion of future research directions.

2 Context-free shuffle grammars

We formalize our extension of CFGs as follows: a CFSG is a quintuple (V, Σ, R, P, S) where V and Σ are finite, disjoint *alphabets* of respectively nonterminal and terminal symbols; R is a finite relation associating nonterminals with strings of nonterminals and terminals, $R \subseteq V \times (V \cup \Sigma)^*$; P is a finite relation associating nonterminals with sets of nonterminals and terminals, $P \subseteq V \times \mathcal{P}(V \cup \Sigma)$; and $S \in V$ is the *starting symbol*. R and P are the rules by which nonterminal symbols may be rewritten to produce (over possibly several rewrites) terminal strings. R gives the traditional production rules of CFGs; the definition of a standard CFG is just a quadruple (V, Σ, R, S) . P gives rules for applying the shuffle operator. For simplicity we require that if $(a_0 \rightarrow a_1 || \dots || a_n) \in P$, then each $a_i \notin \text{dom}(P)$. For example we might have rules

$$S \rightarrow T || U \quad T \rightarrow Wpr \quad U \rightarrow 12345 \quad W \rightarrow mn$$

corresponding to our earlier example, and so formally the sets $R = \{S \rightarrow Wpr, T \rightarrow 12345, W \rightarrow mn\}$ and $P = \{S \rightarrow \{T, U\}\}$. We use upright sans-serif script when writing literal examples, as above, and italicized letters to represent variables or unknown values, with a ranging over single symbols, s ranging over single nonterminals, and u, v over strings of symbols.

Rewriting \Rightarrow

- (\Rightarrow .1) If $a \rightarrow u \in R$, then $a \Rightarrow u$.
- (\Rightarrow .2) If $a \rightarrow \{a_1, \dots, a_n\} \in P$ then we have $a \Rightarrow \{a_1, \dots, a_n\}_a$.
- (\Rightarrow .3) If $(\forall 0 \leq i \leq n) v_i \in \Sigma^*$ and v is a shuffling of the v_i , $v \in v_0 \parallel \dots \parallel v_n$, then $\{v_0, \dots, v_n\}_a \Rightarrow v$.
- (\Rightarrow .4) If $u \Rightarrow u'$, then for any strings u_0, u_1 , $u_0 u u_1 \Rightarrow u_0 u' u_1$.
- (\Rightarrow .5) If $u \Rightarrow u'$, then for any strings u_0, \dots, u_m and nonterminal a , $\{u_0, \dots, u_m, u\}_a \Rightarrow \{u_0, \dots, u_m, u'\}_a$.

Marked rightmost rewriting \Rightarrow_M

- (M.1) If $a \rightarrow u \in R$, then $a \bullet \Rightarrow_M u \bullet$.
- (M.2) If $a \rightarrow \{a_1, \dots, a_n\} \in P$ then we have $a \bullet \Rightarrow_M \{a_1 \bullet, \dots, a_n \bullet\}_a$.
- (M.3) If $(\forall 0 \leq i \leq n) v_i \in \Sigma^*$ and v is a shuffling of the v_i , $v \in v_0 \parallel \dots \parallel v_n$, then $\{\bullet v_0, \dots, \bullet v_n\}_a \Rightarrow_M \bullet v$.
- (M.4) If $u \Rightarrow_M u'$, then for any $u_0 \in (V \cup \Sigma)^*$ and $u_1 \in \Sigma^*$, $u_0 u u_1 \Rightarrow_M u_0 u' u_1$.
- (M.5) If $u_i \Rightarrow_M u'_i$ then $\{u_0, \dots, u_i, \dots, u_n\}_{a_0} \Rightarrow_M \{u_0, \dots, u'_i, \dots, u_n\}_{a_0}$.
- (M.6) For any $s \in \Sigma$, $s \bullet \Rightarrow_M \bullet s$.

Fig. 1. The rewriting and rightmost rewriting relations.

Figure 1 shows the rules for rewriting in CFSGs. The rewriting relation \Rightarrow applies to a pair of strings where the latter is derived from the former by expanding one nonterminal. For rewriting according to the expansions in R we have (\Rightarrow .1). Rules (\Rightarrow .2) and (\Rightarrow .3) treat shuffle expressions. The commas and curly braces introduced in the right-hand side expansion of (\Rightarrow .2) are all interim symbols, not part of V or Σ , which we use as delimiters in non-final rewritten strings. They disappear when we have rewritten their substrings into shuffled terminals by (\Rightarrow .3). Finally rules (\Rightarrow .4) and (\Rightarrow .5) allow rewriting to occur at any position within a string. So in our example grammar we have $T \Rightarrow Wpr \Rightarrow mnpr$ and $U \Rightarrow 12345$. Since $mnpr$ and 12345 are both strings of terminals only, and since $m12np3r45$ is a shuffle of $mnpr$ and 12345 , we have

$$\begin{aligned} S &\Rightarrow \{T, Y\}_S \Rightarrow \{Wpr, Y\}_S \Rightarrow \{Wpr, 12345\}_S \Rightarrow \{mnpr, 12345\}_S \\ &\Rightarrow m12np3r45 . \end{aligned}$$

So $m12np3r45$ is a string in our example grammar's language.

The figure also defines a rightmost marked rewriting relation which we use to support the correctness of our parser. Rightmost marked rewriting adds a position marker \bullet to the structure of rewritten strings, and tracks rewrites to allow them only immediately to the left of the marker. Its first three rules are the same as for unrestricted rewriting \Rightarrow , except for maintaining the marker to the right of possible nonterminals, and to the left of guaranteed terminals.

Moreover (M.5) and $(\Rightarrow.5)$ are also similar, since we allow rightmost rewriting among any of the substrings to be shuffled. In (M.4), marked rightmost rewriting is permissible only when there are no nonterminals textually to the right of the one to be expanded. We write $|u|$ to refer to the erasure of all position markers from u .

Lemma 1. 1. If $u \Rightarrow^* u' \in \Sigma^*$, then $u \bullet \xRightarrow[M]{*} \bullet u'$.
 2. If $u_0 \xRightarrow[M]{*} u_1$ and $|u_0| \neq |u_1|$, then $|u_0| \Rightarrow |u_1|$.

The second clause is clear from erasing the marker, and from the more lenient contexts of \Rightarrow . For the first clause we can use an additional intermediate relation which keeps the contextual restrictions of $\xRightarrow[M]{*}$ but not the markers, to first argue for the reordering of rewrites, and then the addition of markers.

3 Generalized LR-shuffle parsing

Like the standard GLR parser, GLR-S defines an underlying nondeterministic automaton, and optimizes a process for tracking all possible traces through it. We present the underlying nondeterministic automaton in this section, and discuss implementations in Section 4.

Since we have extended the grammar and rewriting language for the shuffle operator, we must revisit the notion of an *item*. As in the traditional case of rules from R , the marker may be at the beginning, the end, or between any two characters of the expansion. We add two sorts of item to the classical notion. Corresponding to a rule in P , we can have items $a \rightarrow \bullet\{a_1, \dots, a_n\}_m$ and $a \rightarrow \{a_1, \dots, a_n\}_m \bullet$. Note that these forms do not have the same syntax as for rewriting; in each form, m is an integer at least as big as n , rather than the original nonterminal. An *initial* item has its marker at the beginning of the right-hand side; a *final* item, at the end. Finally there is a placeholder item ∇a , which we will use to represent a transition to the subtask of recognizing a particular shuffled substring.

As usual when constructing an LR-style parser we use a handle-finding deterministic finite automaton (DFA), derived from a nondeterministic finite automaton (NFA) based on items as states. In our motivating application of plan recognition, lookahead to future actions to be observed is not realistic, so we use an LR(0) handle-finding automaton here, but we expect that for LR(1) or LALR(1) handle-finding we would proceed similarly. In addition to the standard transitions, and to the usual *stations* for the nonterminals themselves [6], from an item $a \rightarrow \bullet\{a_1, \dots, a_n\}_n$ we have a transition labelled a_i to $a \rightarrow \bullet\{a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n\}_n$ for each i , plus an ϵ -transition from any $a \rightarrow \bullet\{\}_n$ to $a \rightarrow \{\}_n \bullet$. We annotate the NDA for our handle finder with

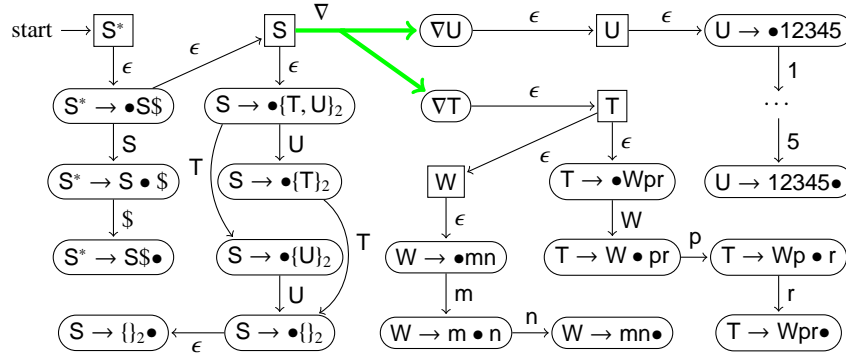


Fig. 2. Nondeterministic handle-finding automata for our example grammar. The hyperedge arising from the shuffle operator is highlighted in green.

hyperedges linking from one state to several. Specifically the graph of states and edges of our extension forms an arc-labeled F-directed hypergraph [1]. Of course the hyperedges have no impact on the operation of an NDA; one does not “run” the handle-finding automaton in any real sense. We use the hyperedges for bookkeeping, translating them to the DFA and generating particular action table entries based on them. Where there is such a hyperedge edge from an item I to items I_i in the NDA, we expect that any state containing I in the DFA would have a similar hyperedge to the least sets containing the I_i . From an item $a \rightarrow \bullet\{a_1, \dots, a_n\}_n$ we add a hyperedge labelled ∇ , a symbol not in the original grammar, to n indirection items $\nabla a_1, \dots, \nabla a_n$; from each indirection item ∇a_i we have an ϵ -transition to the station for a_i . By convention we designate a nonterminal S^* and terminal $\$$ which are not in the original grammar, and take the initial state of the NFA to be the station for S^* . Figure 2 shows the nondeterministic handle-finding automaton for our example grammar, and Figure 3 shows its translation to a DFA.

In non-generalized parsers, we expect that each goto/action table position will have exactly one entry (which might be to fail). This allows the creation of a *deterministic* parser; otherwise the presence of a *defective state* with multiple conflicting entries makes it unclear which of the steps should be followed. The essential insight of Tomita’s GLR parser is that for most grammars, and in particular those which are of practical use, we can efficiently track *all* of the possible actions. So we do not insist that a GLR-S goto/action table be free of conflicts. As usual the table is indexed by state and by symbol; each table entry $[S, a]$ includes a set of instructions, which are empty except as described by these rules:

1. For each $a \rightarrow u\bullet \in S$, add $\text{reduce}(a \rightarrow u\bullet)$ to entry $[S, a']$ for all a' .

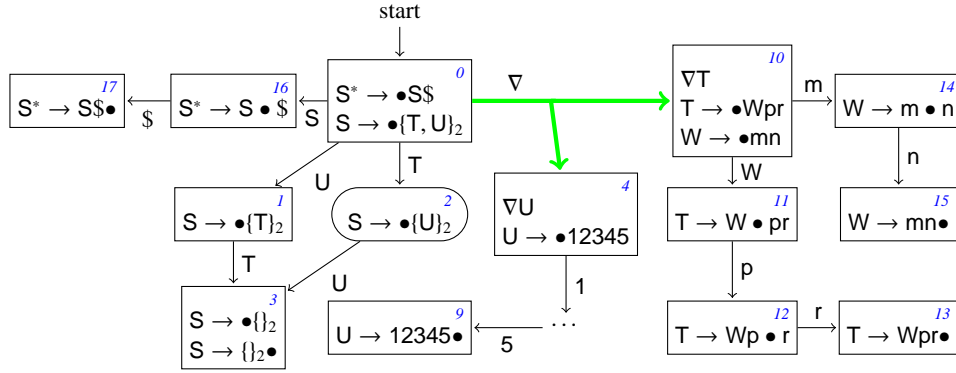


Fig. 3. Conversion of the NDA of Figure 2 into a DFA. Numbers in blue are an arbitrary assignment of a number to each state, which we use in Table 1.

2. For each simple edge $\mathcal{S} \xrightarrow{a} \mathcal{S}'$, add $\text{shift}(\mathcal{S}')$ to entry $[\mathcal{S}, a]$.
3. For each hyperedge $\mathcal{S} \xrightarrow{\nabla} \mathcal{S}_1, \dots, \mathcal{S}_n$ and simple edge $\mathcal{S}_i \xrightarrow{s} \mathcal{S}'_i$ where $s \in \Sigma$, include $\text{shift}(\mathcal{S}_i \rightarrow \mathcal{S}'_i; \mathcal{S}_1, \dots, \mathcal{S}_{i-1}, \mathcal{S}_{i+1}, \mathcal{S}_n)$ to entry $[\mathcal{S}_0, s]$.
4. For any transition by $\$$ to a state containing the accepting item, $\mathcal{S} \xrightarrow{\$} \{\mathcal{S}^* \rightarrow \mathcal{S}\$ \bullet, \dots\}$, we add accept to $[\mathcal{S}, \$]$

The rules for reduce, for shift from a simple edge, and for accept are exactly as for classical CFG parsing. The third rule addresses hyperedges. Through the hyperedges we identify the initial states for recognizing the shuffled strings, but a hyperedge does not correspond to recognizing an actual piece of the input. So the shift operations which initiate recognizing a shuffle derive from two edges, the hyperedge plus a single subsequent edge which actually consumes a piece of input. Table 1 shows the first few rows and columns of the table for our running example. Note that our restriction to P against the referencing further rules in P simplifies Rule 3 here, otherwise we must chase through, and accumulate concurrent terms from, several rules.

With the goto/action table in hand, we can specify a nondeterministic automaton which recognizes strings in our CFSG. Figure 4 presents our algorithm. The runtime state of the parser is a cactus stack — a tree which grows and shrinks at the leaves — of states from the handle finding DFA. For clarity we explicitly mark the nodes which we consider to be stack tops, since we detail the creation of nodes which are only temporarily at a leaf in the tree. We write $\beta : \mathcal{S}$ to name a stack top β containing the state \mathcal{S} . Figure 5 shows the manipulations to the cactus stack as GLR-S recognizes the string m12np3r45 in our example grammar.

	S	T	U	W	m	1	...
0	shift(16)	shift 7	shift 1	–	shift(10 → 14; 4)	shift(4 → 5; 10)	
1	–	shift 3	–	–	–	–	
2	–	–	shift 3	–	–	–	
3	reduce $S \rightarrow \{ \}_2 \bullet$
4	–	–	–	–	–	shift 5	
⋮							

Table 1. Part of the goto/action table for our example grammar. In Row 3, the reduce action appears in every column.

Step 1(c) of the algorithm shows the purpose of the ∇a nodes. The reduce operations in LR parsers rely on the number of states pushed onto the stack for the right-hand side of a rule being the same as the length of that right-hand side. But when we split the stack for shuffled substrings we start the substacks with an initial state that does not correspond to any recognized symbol. The presence of an indirection to ∇a in an item set rectifies this offset. But since the ∇a item would not appear in the state corresponding to a sequential use of a — which would follow an ϵ -link to a 's station instead of to ∇a — we will not disrupt stack operations in non-shuffled cases.

Lemma 2 (Main). *Algorithm 1 accepts a string $u \in \Sigma$ under a grammar with starting symbol S if and only if $S \bullet \xRightarrow[M]{*} \bullet u$.*

Specifically, the manipulations to the cactus stack under the GLR-S algorithm correspond to the reverse of an $\xRightarrow[M]{*}$ sequence:

- Application of a reduce in Step 1 of the algorithm corresponds to a rewrite by Rule (M.1).
- Application of the simple shift operation in Step 2.(b) corresponds to a rewrite by rule (M.6).
- Application of the shuffle-decomposing shift operation of Step 2.(c) corresponds to a rewrite by Rule (M.2), followed by a rewrite by Rule (M.3).

Lemmas 1 and 2 justify the correctness of the GLR-S parser.

Theorem 1 (Correctness). *Algorithm 1 accepts a string $u \in \Sigma^*$ if and only if $S \Rightarrow^* u$.*

4 Necessary optimizations

An important aspect of GLR is the construction of a single structure to represent all current possible parse stacks. To make the representation reasonable, stack

Algorithm 1 (GLR-S parsing) Initially the single state on the stack is the DFA's initial state. For each symbol s of the input string including the end-of-string marker $\$$:

1. For each reducible stack top $\alpha : S$, if reduce $a \rightarrow u\bullet \in S$, then we may choose to reduce that rule:
 - (a) Drop α as a stack top.
 - (b) Pop $|u|$ nodes from α to node α' .
 - (c) If $\alpha' : S'$ contains the indirection node for a , $\forall a \in S'$, pop one additional time to α'' , else take α'' to be just α' .
 - (d) Let $\alpha'' : S_0$, and choose some shift operation $\text{shift}(S'_0) \in [S_0, a]$, or raise an error if there is no possible shift. Create $\beta : S'_0$ with parent α'' .
 - (e) If α'' has other child nodes:
 - i. Then update the other children of α'' to have β as their parent.
 - ii. Else take β as a stack top.
2. Choose a stack top $\alpha : S$ with a stack or accept operation in $[S, s]$ (or reject if there is no such α), and choose one of those operations.
 - (a) If the operation is accept, then the parser accepts the string.
 - (b) If the operation is $\text{shift}(S)$, then create $\beta : S'$ with parent α , and replace α as a stack top with β .
 - (c) If the operation is $\text{shift}(S_0 \rightarrow S'_0; S_1, \dots, S_n) \in$, then create $\beta_i : S_i$ with parent α for each $0 \leq i \leq n$, and moreover create $\beta'_0 : S'_0$ with parent β_0 . Replace α as a stack top with β'_0 and the β_1, \dots, β_n .

After the end-of-string marker $\$$ if we have not accepted the input, then we reject it.

Fig. 4. The nondeterministic GLR-S parsing algorithm.

structure is shared whenever possible. The obvious case is that when two different actions are possible for some stack top, we can simply branch the stack, so that the common stack bottom is shared. Less obvious, but just as important, is the idea that common stack tops should be shared as well: when a node for some state S would be pushed onto n different stacks for one input symbol, we in fact allocate only one single node, with pointers to all of the previous tops-of-stacks. This one single node is taken as the top of a stack, as opposed to maintaining n different stacks. Later, a reduce action can pop nodes past the point where the several stacks join, resulting in several stacks. Tomita dubbed this construction the *graph-structured stack* [12].

We adopt the graph-structured stack for GLR-S, but the midstack mutation performed at Step 1.(e).i of the algorithm complicates its use. In the nondeterministic parser it is simple enough to mutate the middle of a stack, but not all of the stacks superposed in graph-structuring will undergo the same mutations. Consider the fragment of DFA in Figure 6(a). The string ab could be attributed to either T or U , and we have two possible parses, shown in Figure 6(b). We

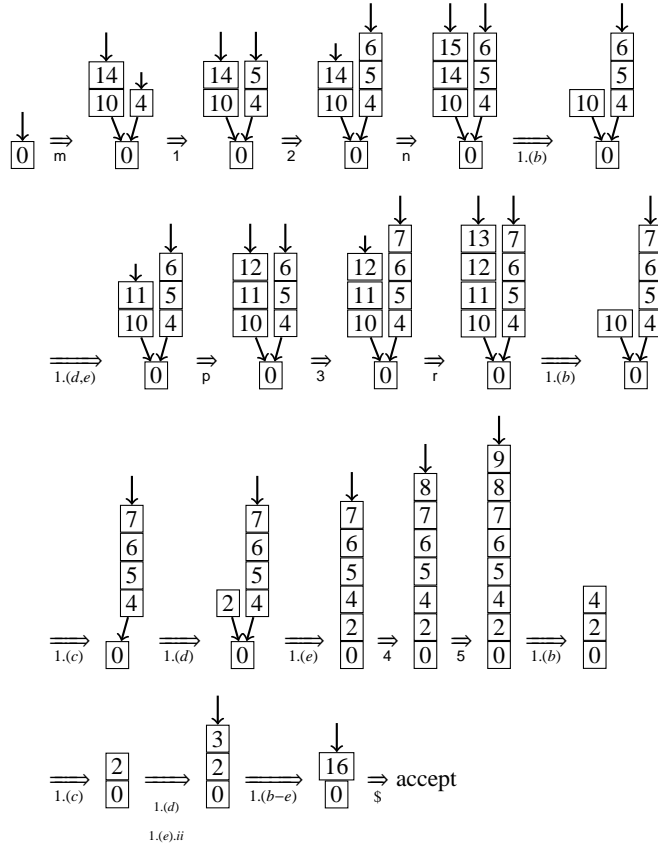


Fig. 5. Parser actions for the string m12np3r45 with our running example.

can easily imagine superposing the stacks resulting after recognizing just **a**: Figure 6(c) shows the combined stacks, with two sets of stack tops depicted in different colors. But the stacks resulting after **b** are more difficult to combine, since they differ not only at their tops but also internally.

To store stacks in such cases we use not one but two graphs, which we distinguish as *upper* and *lower*. The lower graph holds parser states as in Tomita's graph-structured stack, but disconnected at the junctions arising from shuffle operations. The relationship between these graph fragments, as well as the sets of stack tops, are maintained in the upper graph. The upper graph has the structure of an and/or tree, possibly with shared substructure. Or-nodes reflect different possible parses; and-nodes organize recognition of shuffled substrings. To chain together substacks in the lower graph, we bind *controllers* in and-nodes, and

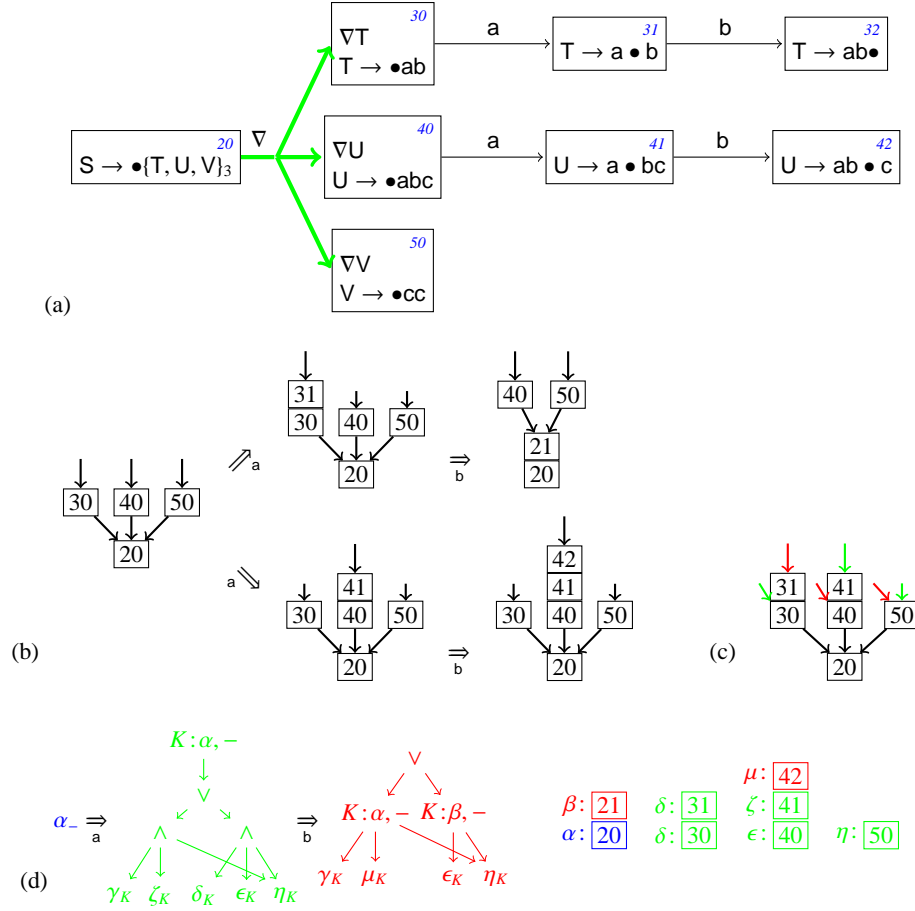


Fig. 6. Example grammar elements and cactus stacks.

reference both a stack top and one of these controllers in each leaf node. The binding of a controller in an and-node associates the controller with both a prior stack top, and a parent controller. So essentially the controllers form a linked tree of stacks from the lower graph, which taken together assemble the cactus stacks of the nondeterministic algorithm. Separating the binding of a controller to a parent stack top on the one hand, from the stack tops in leaf nodes on the other hand, allows us to locally mutate controller bindings in a way that restricts the scope of the effect.

We illustrate the use of controllers in Figure 6(d), with the series of upper graphs to the left, and the lower graph on the right. Note that we label each lower graph node with a Greek letter, and that lower graph nodes take the same

color as the upper graph from which they were added. Before processing input symbol a , we have a single stack top α and a void controller $-$. Starting to parse the shuffled substring, GLR-S creates the controller K , associating it with α as the previous top of stack. Since there are two ways to understand this a , as either the first symbol of T or as the first symbol of U , we see below the new binding two different groups of stack tops. The or node labelled \vee sits below the binding to spare the work of duplicating it; the and-nodes labelled \wedge , which do not bind new controllers, each identify a set of three stack tops reflecting the state of parsing the shuffled strings. When processing the b we find one case where reducing T requires mutating the middle of the stack by pushing a new node above α , and one case where we do not. Reconciling the two does require separate bindings for K , and the resulting pair appears under an or-node. The meaning of a controller reference in a tree leaf is determined with respect to a particular path from the root of the upper graph through various and-nodes with bindings and to the leaf. So the leaves do not change as the upper graph evolves unless a new node is pushed onto them.

For each input symbol we traverse the upper graph, constructing a new upper graph while reusing as much of the previous graph as possible. When traversing an or-node we discard any children for which there is no way to advance with the next input. For an and-node we require exactly one of its children to advance for the new input symbol; if more than one can advance, then we will have a disjunction for each possible evolving child, with the other child graphs in each case unchanged. We apply the usual shift and reduce operations at leaf nodes. It is when a reduce operation exhausts a stack via Step 1(c) of the algorithm that we update the controller binding, possibly branching to multiple bindings of the controller as in the example. We can optimize the traversal and preserve sharing of subgraphs by caching the map from old to new upper graph structures, traversing a shared subgraph once only.

5 Conclusion

We have presented an extension to GLR parsing for languages generated by context-free grammars enriched with the shuffle operator, and discussed its correctness and its efficient implementation. We conclude with two avenues of future work.

Along with the GLR algorithm Tomita gave an approach for efficiently representing a *parse forest*, a collection of parse trees, of all possible derivations, and we have left the adaptation of this technique to GLR-S parsing to future work. An efficient representation of these parse forests is quite relevant to the artificial intelligence applications of this work. Capturing the parse forest rep-

resents the difference between goal recognition, where only the top-level intentions of an actor are determined, and plan recognition, where in addition to the goal a detailed plan is constructed. The cost of retaining plans is not inconsiderable, and efficient recognition of full plans remains an area of active study [7].

A formal assessment of the worst- and average-case complexity of GLR-S parsing also remains to be done. However, we have implemented a prototype of a plan recognizer based on GLR-S parsing, and our preliminary testing suggests that it improves considerably over past approaches, with near-linear performance for randomly generated libraries which reflect common use cases.

References

1. Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Ngyyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201, 1993.
2. Christopher W. Geib. Delaying commitment in plan recognition using combinatory categorical grammars. In *Proc. 21st Int. Joint Conf. on Artificial Intelligence*, 2009.
3. Christopher W. Geib and Robert P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 117(11):1101–1132, July 2009.
4. Christopher W. Geib, John Maraist, and Robert P. Goldman. A new probabilistic plan recognition algorithm based on string rewriting. In *Proc. 18th Int. Conf. on Automated Planning and Scheduling*, pages 91–98, September 2008.
5. Robert P. Goldman, Christopher W. Geib, and Christopher A. Miller. A new model of plan recognition. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pages 245–254, July 1999.
6. Dick Grune and Criel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, 2008.
7. Reuth Mirsky and Ya’akov Gal. SLIM: Semi-lazy inference mechanism for plan recognition. In *Proc. 25th Int. Joint Conf. on Artificial Intelligence*, July 2016.
8. David V. Pynadath and Michael P. Wellman. Probabilistic state-dependent grammars for plan recognition. In *Proc. 16th Conf. on Uncertainty in Artificial Intelligence*, pages 507–514. Morgan Kaufmann Publishers Inc., 2000.
9. Antonio Restivo. The shuffle product: New research directions. In *Proc. 9th Int. Conf. on Language and Automata Theory and Applications*, pages 70–81, March 2015.
10. Martin Sulzmann and Peter Thiemann. Derivatives for regular shuffle expressions. In *Proc. 9th Int. Conf. on Language and Automata Theory and Applications*, March 2015.
11. Martin Sulzmann and Peter Thiemann. Derivatives and partial derivatives for regular shuffle expressions. *Journal of Computer and System Sciences*, submitted.
12. Masaru Tomita. An efficient context-free parsing algorithm for natural languages. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 756–764, August 1985.
13. Masaru Tomita, editor. *Generalized LR Parsing*. Kluwer Academic Publishers, 1991.
14. Marc Vilain. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proc. 8th Nat. Conf. on Artificial Intelligence*, pages 190–197, 1990.
15. Marc Vilain. Deduction as parsing: Tractable classification in the KL-ONE framework. In *9th Nat. Conf. on Artificial Intelligence*, pages 464–470, July 1991.